SEUPD@CLEF: Team CLOSE

Farzad **Shami**[1], Mirco **Cazzaro**[1], Marco **Martinelli**[1], Nicola Boscolo **Cegion**[1], Seyedreza **Safavi**[1] and Gianluca **Antolini**[1]

[1]*University of Padua, Italy*

Abstract

This paper presents the work of the CLOSE group, a team of students from the University of Padua, Italy, for the 2023 CLEF LongEval Task 1. Our work involved developing an information retrieval system that can handle changes in data over time while maintaining high performance. We first introduce the problem as stated by CLEF and then describe our system, explaining the different methodologies we implemented. We provide the results of our experiments and analyze them based on the choices we made regarding various techniques. Finally, we propose potential avenues for future improvement of our system.

Keywords

Information Retrieval, Search Engines, Longitudinal Evaluation, Model Performance

1. Introduction

Recent research has shown that the performance of information retrieval systems can deteriorate over time as the data they are trained on becomes less relevant to current search queries. This problem is particularly acute when dealing with temporal information, as web documents and user search preferences evolve over time. In this paper, we propose a solution to this problem by developing an information retrieval system that can adapt to changes in the data over time, while maintaining high performance.

Our approach involves using the training data provided by Qwant search engine, which includes user searches and web documents in both French and English. We believe that this data will enable our system to better adapt to changes in user search behavior and the content of web documents.

The remainder of this paper is organized as follows: Section **??** describes our approach in more detail, including the different techniques we used. Section **??** explains our experimental setup, including the datasets and evaluation metrics we used. Section **??** presents our main findings and analyzes them based on the choices we made regarding various techniques. Finally, Section **??** summarizes our conclusions and outlines potential avenues for future work.

---

2. Methodology

2.1. General Overview of Our IR System

In the development of our **IR! (IR!)** system, we followed the traditional Y model suggested by Apache Lucene [**?**], represented in the figure below.
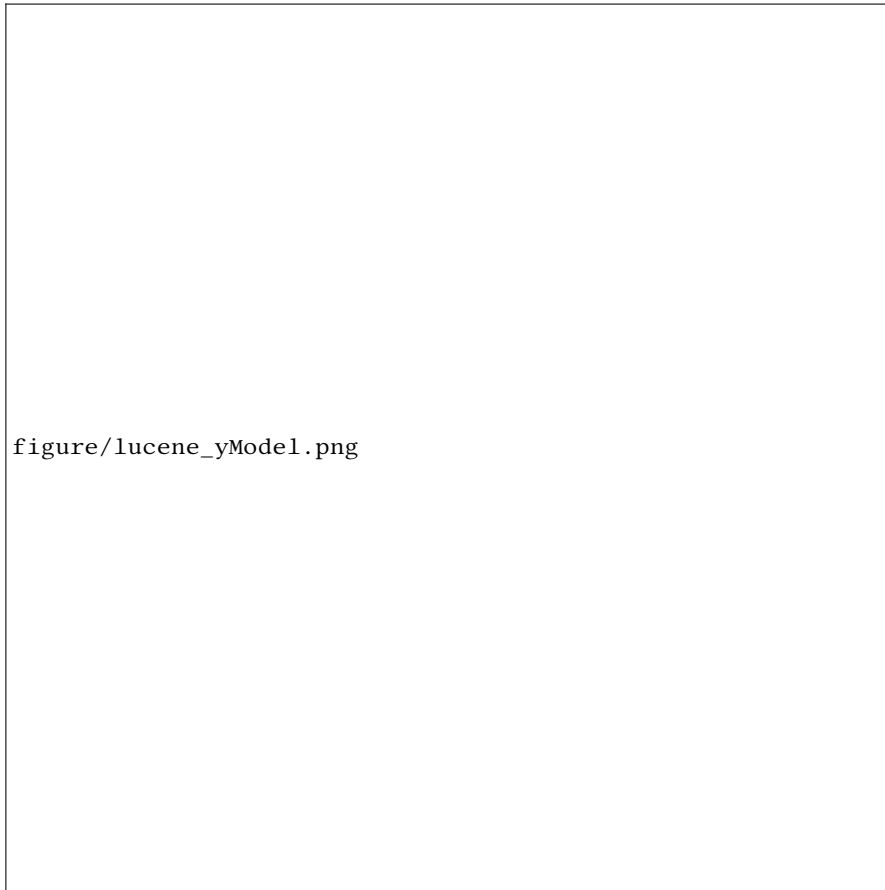

figure/lucene_yModel.png

Figure 1: Apache Lucene Y Model.

The workflow of our system, starting from the train collection provided by **CLEF! (CLEF!)**, is as follows:

1. **Parsing**: The first phase consists of parsing the documents in the collection, which is a pre-processing operation performed to clean them from unnecessary noises. Since the collection is composed of web pages, the documents contain many leftovers like JavaScript scripts, HTML and CSS codes, HTTP and HTTPS URIs, and so on. The purpose of this phase is to ease the processing performed in the following phases.

2. **Indexing**: Each parsed document is then analyzed and indexed keeping only the necessary information. Indexed documents are composed of two fields: an *id* field, containing the

identifier of the document in the collection, and a *content* field, containing the entire body of the document cleaned by the parsing and indexing phases.

3. **Query Formulation**: Topics are then parsed using the same analyzer used for documents, and used to formulate queries. For each topic, together with the already provided query, around 15 others are computed and used altogether for searching relevant documents.

4. **Re-ranking**: The retrieved documents are re-ranked combining the scores obtained with two different approaches which compute the similarity of the document to the query given in the topic.

figure/CLOSE_IR_Workflow (2).pdf

Figure 2: Workflow of the IR system implemented by *CLOSE*.

2.2. Project Structure

The project is developed mainly in *Java*, with the addition of some *Python* scripts for performing the expansion of the queries and for the re-ranking of the retrieved documents for each topic. The overall structure is as follows:

- **experiment**: This folder contains the results of the indexing obtained with different approaches, stored in different sub-folders.
- **python_scripts**: This folder contains the above-mentioned *Python* scripts, together with other *txt* and *JSON* files that these scripts use.
- **src/main/java**: This folder contains all the *Java* classes used for the implementation of the **IR!** system, divided into the following packages:
    - **parser**: This package contains the classes involved in the pre-processing of the document, i.e., the parsing phase.
    - **analyzer**: This package contains the classes used for analyzing the parsed documents and topics of the collection.
    - **indexer**: This package performs the indexing of the documents processed by the analyzer.
    - **searcher**: Once the documents are indexed, the queries contained in the topics of the collection are parsed and then expanded in this package. Relevant (or assumed to be so) documents are retrieved and scored.
    - **utils**: This package contains general utility classes, one of which performs the re-ranking of the documents retrieved by the *searcher*.

A schema of the overall project structure is the following:

```
/code
|------ experiment
|------ python_scripts
|------ src
        |------ main
                |------ java
                        |------ parser
                        |------ analyzer
                        |------ indexer
                        |------ searcher
                        |------ utils
```

2.3. Class Diagram

figure/Classes_diagram_white.pdf

Figure 3: Diagram of the classes implemented by *CLOSE* **IR!** system

2.4. Parser

As stated before, the documents in the collection provided by the **CLEF!** *Long Eval LAB 2023* are essentially the corpus of web pages, to better represent the nature of a web test collection. From this, the need for performing a pre-processing phase of parsing the documents before analyzing and indexing them arises. In this phase, the documents are cleaned from all the residuals of codes not useful for our purposes. We extended Apache Lucene's *DocumentParser* abstract class by creating a custom *ClefParser*, which contains many functions for removing sundry types of noises that can be present in documents. This was the result of the trial and error approach we adopted for implementing this class:

- We started by reading a large statistical sample size of the documents in the collection to decide which types of noises needed to be removed.
- Then we implemented the parser and ran it.
- The results of the parsing were stored, and a sample of the parsed documents was analyzed to start this procedure again.
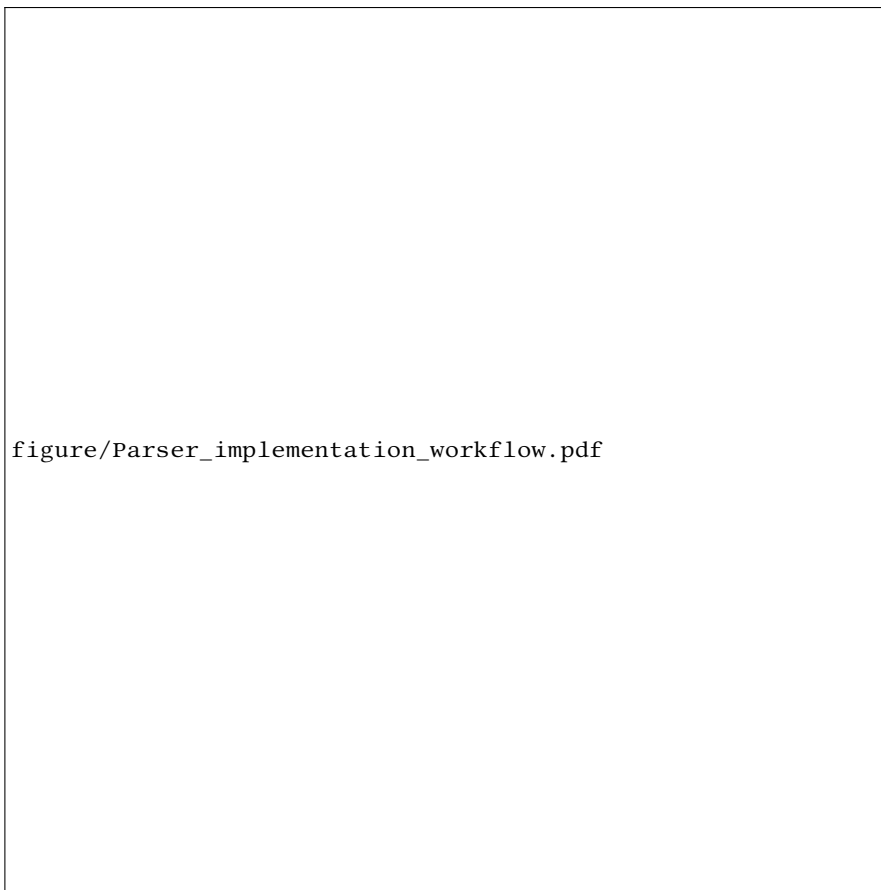

Figure 4: Workflow of the parser implementation.

The types of noises we tried to remove are the following:

- *JavaScript* scripts
- *HTTP* and *HTTPS* URIs
- *HTML* tags and *CSS* stylesheets
- *XML* and *JSON* codes
- Meta tags and document properties
- Navigation menus
- Advertisements
- Footers
- Social media handlers
- Hashtags and mentions

We also identified some patterns of words and symbols to remove:

- Two words separated by an underscore, like *word1_word2*
- Two words separated by a colon, like *word1:word2*
- Two words separated by a point, like "*word1.word2*"

We used *Regular Expressions* to define the patterns to identify and remove. The code below shows the *removePatterns* function, used by the parser for removing the patterns mentioned above together with the *HTTP* and *HTTPS* URIs.

```java
// Function to remove patterns like "word1_word2", "word1.word2",
//    "word1.word2", and HTTP/HTTPS URIs from a string
public String removePatterns(String input) {

    // Define regular expression pattern to match the desired patterns
    Pattern pattern = Pattern.compile("(\\w+)[_.:](\\w+) |
        (https?://[\\w-]+(\\.[\\w-]+)+([\\w.,@?^=%&:/~+#-]*[\\w@?^=%&/~+#-])?)
        ");

    // Replace all matched patterns with a space character
    Matcher matcher = pattern.matcher(input);
    String output = matcher.replaceAll(" ");

    return output;
}
```

After different tries, we decided to use only the function above for removing noises, which gave an improving in MAP of +0.5%, together with another one used to remove *JavaScript* codes, which by itself gave +3% on MAP.
The code of the *JavaScript* parser function is the one below:

```java
public String removeJavaScript(String documentBody) {
    // Get the body of the document.
    StringBuilder bodyBuilder = new StringBuilder(documentBody);
```

```java
        //compiles regular expression for JS and all caps text
        Pattern jspattern=Pattern.compile("function.(.)[{]");

        int start=0;
        int end=0;
        //removes all <scripts>
        while((start=bodyBuilder.indexOf("<script", start))!=-1){
            if((end=bodyBuilder.indexOf("script>", start))!=-1){
                end=end+7;
                bodyBuilder.replace(start, end, "");
                continue;
            }
            if((end=bodyBuilder.indexOf(">", start))!=-1){
                end++;
                bodyBuilder.replace(start, end, "");
                continue;
            }
            start++;
        }

        //removes JS
        Matcher m = jspattern.matcher(bodyBuilder);
        while(m.find()){
            start=m.start();
            int count=1;
            for(int i=start; i<bodyBuilder.length(); i++){
                if(bodyBuilder.charAt(i)=='{'){
                    count++;
                    continue;
                }
                if((bodyBuilder.charAt(i)=='}')){
                    if(--count==0){
                        bodyBuilder.replace(start, i, "");
                        m = jspattern.matcher(bodyBuilder);
                        break;

                    }
                }
            }
        }

        return bodyBuilder.toString();
    }
```

The structure of the parsed document is defined in the *ParsedTextDocument* class, and it is composed of just two fields, as provided by **CLEF!**:

1. *id*: the identifier of the document,
2. *body*: the (parsed) content of the document.

The constructor of *ParsedTextDocument* is the following:

```
/**
 * Creates a new parsed document.
 *
 * @param id the document identifier.
 * @param body the document body.
 */
public ParsedTextDocument(final String id, final String body)
```

Inside it, multiple controls about the validity and integrity of the parameters are performed, then an object of the class is instantiated.

The class also contains the following utility methods:

- *getIdentifier*, which returns the (unique) document identifier,
- *getBody*, which returns the body of the document,
- *toString*, which returns a *String* representation of the document,
- *hashCode*, which returns the hash code of the document identifier,
- *equals(Object o)*, which returns true if the two objects have the same identifier.

2.5. Analyzer

The Analyzer is responsible for analyzing the extracted documents and preparing them for the Indexing and Searching phases. It does so by combining a series of techniques of text processing such as tokenization, stemming, stopword removal, and many more.

We extended Apache Lucene's Analyzer abstract class by creating a custom CloseAnalyzer, which is fully customizable using its parameters that can be chosen when creating an instance of the class. This is because we tried different settings and approaches to maximize the results and kept all the possible variations as optional settings. This CloseAnalyzer is passed as a parameter and then used by the DirectoryIndexer and the Searcher.

The constructor of CloseAnalyzer accepts the following parameters:

- **tokenizerType**: used to choose between three standard Lucene tokenizers: Whitespace-Tokenizer, LetterTokenizer, and StandardTokenizer.
- **stemFilterType**: the possible choices for the stemming types are four standard Lucene filters: EnglishMinimalStemFilter, KStemFilter, PorterStemFilter, and FrenchLightStemFilter. We also tried using FrenchMinimalStemFilter and a custom filter called LovinsStemmerFilter based off a LovinsStemmer implementation but decided to keep them commented as they didn't improve the results.
- **minLength** and **maxLength**: these are integers that simply specify the minimum and maximum length of a token, applying Lucene's LengthFilter.
- **isEnglishPossessiveFilter**: specifies whether to use Lucene's EnglishPossessiveFilter or not. Of course, this can be useful when operating with the English dataset.
- **stopFilterListName**: with this parameter, it's possible to insert the path of an eventual word stoplist .txt file located in the *resources* folder. To do this we use Lucene's StopFilter

and a custom class called AnalyzerUtil that uses a *loadStopList* method to read and load all the stoplist words from the specified file. The stoplists we created are based on the standard ones but modified after inspecting the index with the Luke tool. We have lists of different lengths and different ones for French and English.

- **nGramFilterSize**: if specified, this parameter is used to define the size of the n-grams to be applied by Lucene's NGramTokenFilter.
- **shingleFilterSize**: similar to the previous one, if used, this integer number indicates the shingle size to be applied by Lucene's ShingleFilter that allows the creation of a combination of words.
- **useNLPFilter**: this boolean allows the use of Lucene's OpenNLPPPOSFilter for Part-Of-Speech Tagging and of a custom class called OpenNLPNERFilter for Named Entity Recognition. To load the .bin models, which are located in the *resources* folder, we use two methods from AnalyzerUtil: *loadPosTaggerModel* and *loadNerTaggerModel.*
- **lemmatization**: specifies whether to use Lucene's OpenNLPLemmatizerFilter by loading a .bin model file in the *resources* folder using AnalyzerUtil *loadLemmatizerModel* function.
- **frenchElisionFilter**: we applied this only when using the French dataset by adding Lucene's ElisionFilter with an array of the following characters: 'l', 'd', 's', 't', 'n', 'm'.

On top of this, a LowerCaseFilter is always applied.

We also tried Lucene's *ASCIIFoldingFilter* and SynonymGraphFilter. For the second one, only for the French Dataset, we used a SynonymMap based on a .txt file containing French synonyms. After different trials with different variations of the parameter, the following is the instance of the CloseAnalyzer we used:

```
final Analyzer closeAnalyzer = new
    CloseAnalyzer(CloseAnalyzer.TokenizerType.Standard, 2, 15, false,
    "new-long-stoplist-fr.txt", CloseAnalyzer.StemFilterType.French, null,
    null, false, false, true);
```

We have opted for the French dataset and by doing so we have the StandardTokenizer, 2 and 15 as minimum and maximum token length, we use frenchElisionFilter, FrenchLightStemFilter, and a list of 662 French words as a stoplist. We didn't use any of the other parameters.

2.6. Indexer

The indexer is in charge of calling the *Parser* and the *Analyzer*, taking the document processed by these two and indexing them as an object of the *ParsedTextDocument* defined in the *Parser*. The class which performs the document processing mentioned above is *DirectoryIndexer*, whose constructor is the following:

```
public DirectoryIndexer(final Analyzer analyzer, final Similarity
    similarity, final int ramBufferSizeMB,
              final String indexPath, final String docsPath, final
                  String extension,
              final String charsetName, final long expectedDocs,
              final Class<? extends DocumentParser> dpCls) {
```

It takes the following inputs:

- **analyzer**: the *Analyzer* to be used, in our case *CloseAnalyzer*;
- **similarity**: the *Similarity* to be used by the *Analyzer*;
- **ramBufferSizeMB**: the size in megabytes of the RAM buffer for indexing the documents;
- **indexPath**: the directory where to store the index;
- **docsPath**: the directory from which the documents of the collection have to be read;
- **extension**: the extension of the files in the collection to be indexed;
- **charsetName**: the name of the charset used for encoding documents;
- **expectedDocs**: the total number of documents expected to be indexed;
- **dpCls**: the class of the *DocumentParser* to be used, in our case *ClefParser*.

The function which actually performs the storing of the index is:

```
public void index()
```

Its main functionality is shown below:

```java
// Create a stream of parsed documents from the file with the given
    Parser class(dpCls)
DocumentParser.create(dpCls, Files.newBufferedReader(file,
    cs)).forEach(pd -> {
  Document doc = new Document();
  ParsedTextDocument ptd = (ParsedTextDocument) pd;

  // add the document identifier
  doc.add(new StringField(ParsedTextDocument.Fields.ID,
      ptd.getIdentifier(), Field.Store.YES));

  // add the processed document content
  doc.add(new BodyField(ptd.getBody()));
  try {
    writer.addDocument(doc);
  } catch (IOException e) {
    e.printStackTrace();
  }

  docsCount++;

  // print progress every 10000 indexed documents
  if (docsCount % 10000 == 0) {
    System.out.printf("%d document(s) (%d files, %d Mbytes) indexed in
        %d seconds.%n",
    docsCount, filesCount, bytesCount / MBYTE,
    (System.currentTimeMillis() - start) / 1000);
  }
});
```

which reads, indexes, and stores the processed documents.

The indexer contains some printouts used to see the progress of the indexing during the executions, such as the number of indexed documents, the time elapsed, if the number of documents indexed is less than expected, and some other eventual warning that can come out during this operation.

The overall workflow, starting from the documents collection and ending with having the indexed documents stored, is the following:
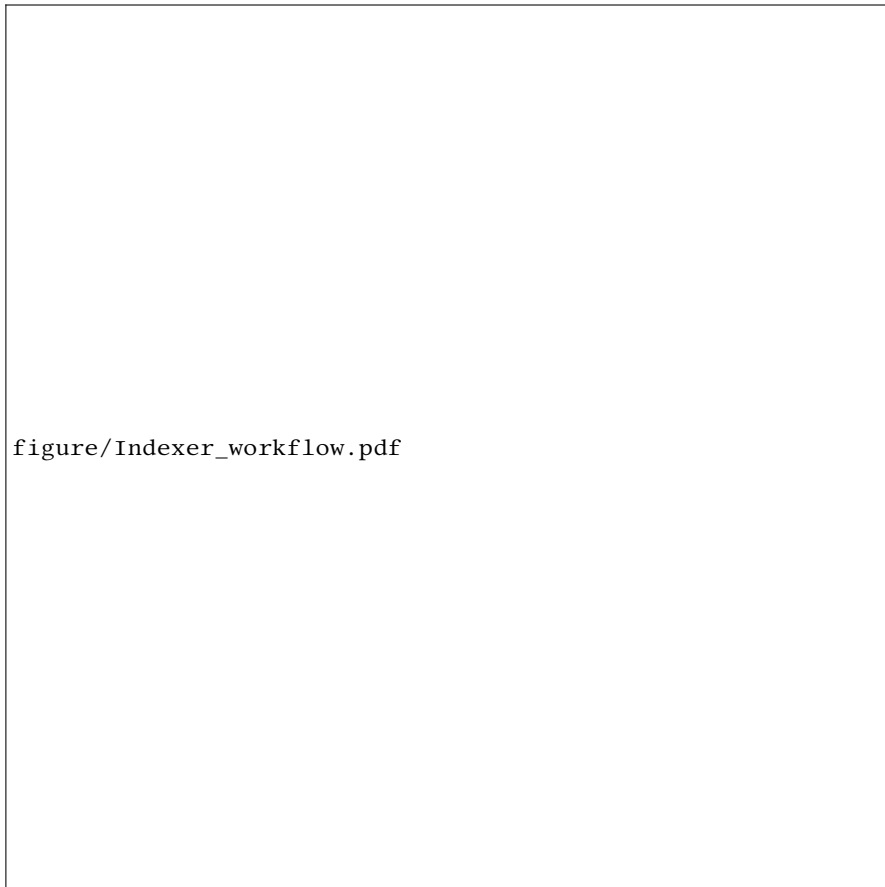
```
figure/Indexer_workflow.pdf
```

Figure 5: Workflow of the indexer.

2.7. Searcher

The purpose of the Searcher is to search through the indexed documents to retrieve relevant information based on user queries after analyzing them and to return a ranked list of documents that match the user's information needs.

Our implementation does so by accepting the following parameters:

- **analyzer**: in this case, an instance of CloseAnalyzer.

- **similarity**: we decided to opt for the BM25Similarity function with the parameters $k1$ and b tuned at 1.2 and 0.90.
- **Run options**: there are parameters for the index path, the topics path, the run path and the run name, the number of the expected topic (in our case 50), and the maximum number of documents retrieved (in our case 1000).
- **useEmbeddings**: a boolean value to decide whether to use embeddings or not. If it is set to true we don't use the similarity function. In our implementation, we opted to not use them.
- **reRankModel**: this is the type of model used to do a Re-Ranking on the retrieved documents. In our case, we use a model called *all-MiniLM-L6-v2*, explained in the following subsection. If the parameter is set to null, no model is used and the documents are scored normally.

2.7.1. Document Re-Ranking

This is the process of ranking the documents retrieved by the search function of the Searcher a second time, using, in our case, a sorting algorithm done with the help of *all-MiniLM-L6-v2* sentence-transformer model, coming from a Python framework called *Sentence Transformers* available at https://huggingface.co/sentence-transformers.
It works in the following way: in the constructor of the Searcher the Re-Ranker is initialized and a predictor is created to perform inference. At the end of the search function, we call a *sort* function that, using the predictor, creates embeddings for the documents and, for each query, calculates the similarity between the query and the documents that are finally multiplied by the actual document's score. The documents are then sorted by looking at these new scores.

2.7.2. Query Expansion

When running the search function, one of the first actions performed is getting the new queries generated with query expansion. We created a Python script that, given the *train.trec* file, generates all the expanded terms for each query and stores everything in a .json file called *result*. Queries are reformulated to increase the probability of matches.

2.7.3. Query Boosting

Query boosting is a technique used to assign greater relevance to certain query terms or queries. We tried the following approach that seemed to improve the overall results: when building the queries in the search function of the Searcher, for each query, a BooleanQuery is built in the following way: after getting the query expansions, each of them is added to the BooleanQuery with the clause *SHOULD* (meaning that at least one of them must be satisfied) and a main query is added with the clause *MUST*, indicating that it must be satisfied. This main query is boosted using Lucene's BoostQuery, with a boost value tuned at 14.68 multiplied by the number of expansions.

3. Experimental Setup

3.1. Collections

We developed our model using a collection of 1,593,376 documents and 882 queries provided by Qwant search engine, available at https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-5010.

The collection contains information about user web searches and actual web pages corpora. The data is originally all in French but, for both queries and documents, an English translation is provided.

3.2. Evaluation Measures

To measure the effectiveness of our Information Retrieval system we used the *trec_eval* executable by testing it with the resulting runs produced by the model and its different configurations.

We tracked improvements of the following evaluation measures generated by *trec_eval*:

- **num_ret**: number of documents retrieved for a given query.
- **num_rel**: number of relevant documents for a given query.
- **num _rel_ret**: number of relevant documents retrieved for a given query.
- **map**: Mean Average Precision, a measure of the average relevance of retrieved documents across all queries. Its range is 0-1 and 1 indicates that all relevant documents are ranked at the top of the result list.
  It's calculated as follows: $MAP = \frac{1}{N}\sum_{i=1}^{N} AP_i$ where N is the total number of queries and AP_i is the average precision of $query_i$ and calculated as: $AP = \frac{1}{RD}\sum_{k=1}^{n} P(k)r(k)$ with RD the number of relevant documents for the query, n the number of total documents, $P(k)$ the precision at k and $r(k)$ the relevance of the kth retrieved document (0 if not relevant, 1 otherwise).
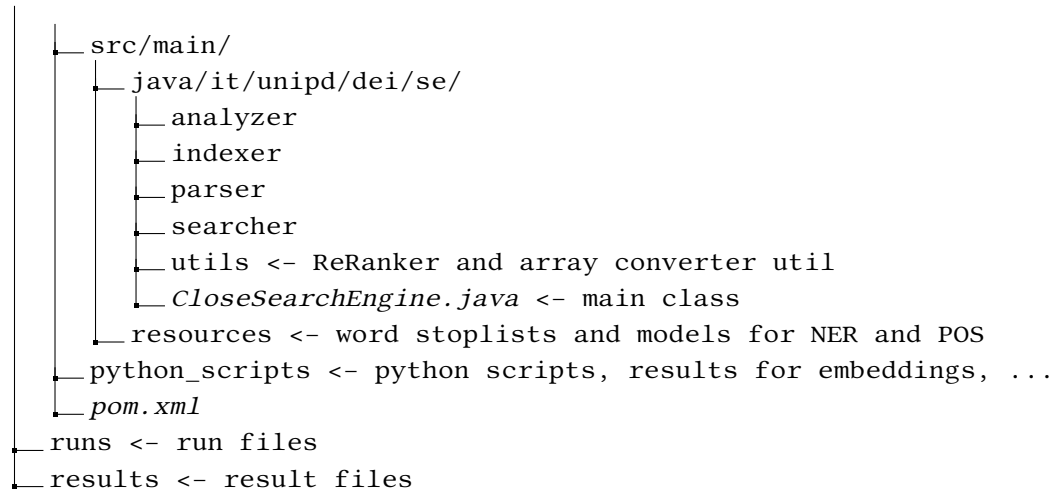 A high MAP score indicates that the model effectively retrieves relevant documents for a wide range of queries.
- **rprec**: R-Precision is the precision score computed at the rank corresponding to the number of relevant documents for a given query.
- **p@5** and **p@10**: Precision at 5 and at 10 is the precision computed at the top 5 and 10 retrieved documents for a given query. They are calculated as follows: $P(5) = \frac{1}{5}\sum_{k=1}^{5} r(k)$ and $P(10) = \frac{1}{10}\sum_{k=1}^{10} r(k)$ with $r(k)$ the relevance of the kth document. These two values can be useful to understand if the system retrieves relevant documents early in the result list, making it easier for the user to find the information needed.

3.3. Git Repository

The git repository of the project can be found at https://bitbucket.org/upd-dei-stud-prj/seupd2223-close/src/master/ and it is organized as follows:

```
/
└── code/
```

```
    │   └─ src/main/
    │       ├─ java/it/unipd/dei/se/
    │       │   ├─ analyzer
    │       │   ├─ indexer
    │       │   ├─ parser
    │       │   ├─ searcher
    │       │   ├─ utils <- ReRanker and array converter util
    │       │   └─ CloseSearchEngine.java <- main class
    │       └─ resources <- word stoplists and models for NER and POS
    ├─ python_scripts <- python scripts, results for embeddings, ...
    └─ pom.xml
├─ runs <- run files
└─ results <- result files
```

3.4. System Hardware

For the most time during the development of the system, every member of the group ran the model on its own system but after implementing the deep learning techniques we decided to switch and start running the tasks using GPU to improve time performances. The following are the specifics of the machines used after switching to computing also with GPU:

- CPU: Intel® Core™ i9-12900H 12th generation
- GPU: NVIDIA RTX™ A 2000 4GB GDDR6
- RAM: 16 GB SO-DIMM DDR5 4800MHz
- SSD: 512 GB M.2 2280 PCIe Gen4 TLC Opal

- CPU: Ryzen 7 1700 overclocked at 3.8GHz
- GPU: RTX 3070 TI 8GB GDDR6X
- RAM: 16GB DDR4 3000MHz
- SSD: Samsung evo 970 250GB

4. Results and Discussion

In this section we provide some of the most relevant results we got during the development phase. We are considering five principal milestones that, within many different trials, led us to improve significantly our MAP score and the overall number of relevant documents actually retrieved.

| metric	run1	run2	run3	run4	run 5
num_q	663	663	669	669	667
num_ret	654909	655684	658446	658347	657903
num_rel	2571	2571	2611	2611	2600
num_rel_ret	1914	2084	2182	2191	2232
map	0.1553	0.1875	0.2022	0.2029	0.2351
gm_map	0.0178	0.0348	0.046	0.0459	0.0629
Rprec	0.1276	0.1571	0.1697	0.1628	0.2022
bpref	0.3331	0.3626	0.3734	0.3761	0.3861
recip_rank	0.261	0.3037	0.3287	0.3281	0.3945
iprec_at_recall_0.00	0.2775	0.3242	0.3499	0.3526	0.4182
iprec_at_recall_0.10	0.2771	0.3236	0.3494	0.3523	0.4175
iprec_at_recall_0.20	0.2577	0.3076	0.3324	0.337	0.3927
iprec_at_recall_0.30	0.2113	0.2523	0.2679	0.2743	0.3236
iprec_at_recall_0.40	0.1732	0.2152	0.2316	0.2323	0.2716
iprec_at_recall_0.50	0.1614	0.2036	0.2193	0.2192	0.2533
iprec_at_recall_0.60	0.133	0.1636	0.1786	0.1758	0.2014
iprec_at_recall_0.70	0.1061	0.1281	0.1402	0.1402	0.1552
iprec_at_recall_0.80	0.0892	0.1091	0.1178	0.12	0.1295
iprec_at_recall_0.90	0.0742	0.0896	0.0956	0.0992	0.1033
iprec_at_recall_1.00	0.0741	0.0895	0.0954	0.0986	0.1028
P_5	0.1222	0.1475	0.1599	0.1614	0.1874
P_10	0.0973	0.1192	0.1296	0.1312	0.1432
P_15	0.0797	0.0965	0.1049	0.1068	0.1135
P_20	0.0668	0.0805	0.088	0.0891	0.0935
P_30	0.0519	0.0616	0.0661	0.0668	0.0703
P_100	0.0206	0.0236	0.0256	0.0258	0.0268
P_200	0.0116	0.0132	0.014	0.0141	0.0148
P_500	0.0053	0.0059	0.0061	0.0061	0.0064
P_1000	0.0029	0.0031	0.0033	0.0033	0.0033
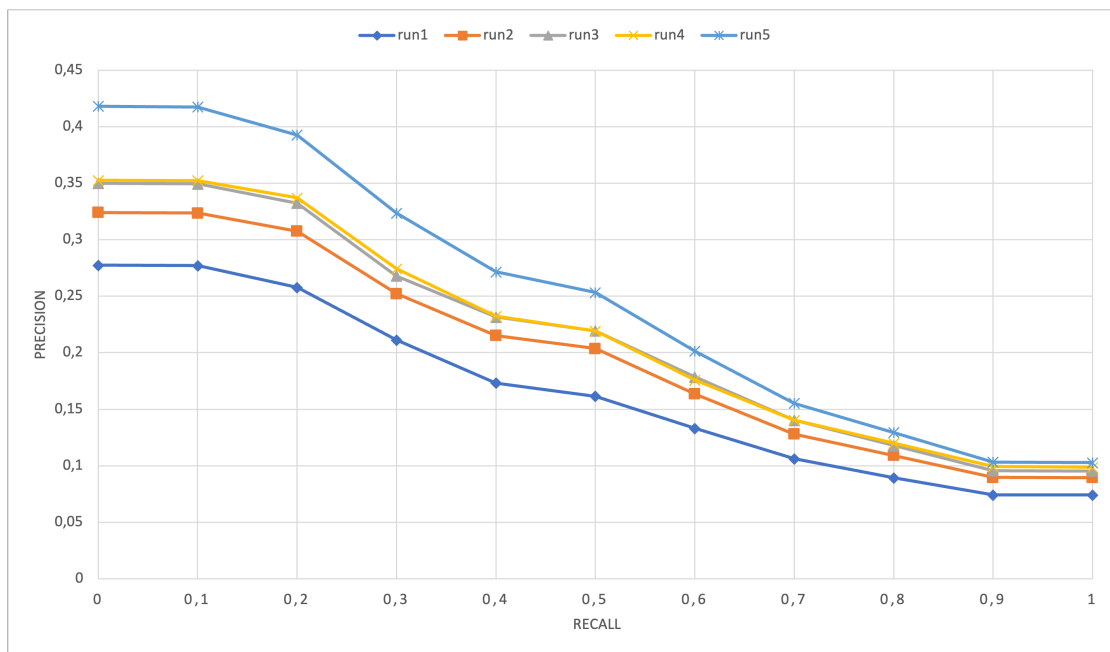
Figure 6: Recall and precision graph

First of all, given that we were provided with two different version of the same document's *corpora*, our first idea was to try with english version.

We noticed that the best combination of basic IR tools were to use the Porter stemmer, a lenght filter from 1 to 10, and a list of stopword composed by some standard terms and more from the top 600 extracted from the index. The very first big milestone, that helped us to increment the MAP of around 3.5 points, was the JavaScript code parser, since we noticed by inspection that many documents were having code inside.

Always by inspecting some documents and queries, and also considering that the original collection was the french version (translated then in english), we observed that the translation was very poor: by switching to french with just parsing the JS code and some other minor parsers, without even using an adequate stoplist and a correct stemmer for the french language, the MAP was increasing by 5 percentual points; 2 more points were achieved with a stop list built for french in the same way we did for previously for english, the *FrenchLightStemFilter* as stemmer, and moving the lenght filter from 2 to 15 (as french probabily usually has longer words). We tried some NLP techniques for english, in particular using *Part-Of-Speech* techniques, to see if there were improvements, and in case apply it to our main implementation for french with an appropriate model, but results were not interesting, and also the computing time were definitely too costly. Another approach we tried and that carried an improvement was to use *Query expansion*: we used some generative text models to expand our queries: we then decided to weight different query scores by boosting the original one linearly with respect to the number of expansion used, and without boosting the expansion: this carried to us an extra MAP point. We tried to combine different similarities rather than using the classic BM25Similarity: we tried to use the Lucene MultiSimilarity, that allows to combine equally the score of two or more similarity scores, but it does not allow to tune the weights. Then, we tried to reimplement the MultiSimilarity class with tuning options, but results were always lower than the standard BM25Similarity. Some minor improvements came up by fine tuning the document-length normalization b, and the term frequency component $k1$ parameters of the BM25. The last main implementation we did was to use some *Reranking* techniques: Lastly, some minor addings were setted on the analyzer by implementing the Lucene ElisionFilter (for french), that aims to remove apostrophed articles and prepositions from tokens.

5. Conclusions and Future Work

Provide a summary of what are the main achievements and findingsssS.

Discuss future work, e.g. what you may try next and/or how your approach could be further developed.

References